

Flexible Cross-Domain Event Delivery for Quality-Managed Multimedia Applications

CHRISTIAN POELLABAUER

University of Notre Dame

and

KARSTEN SCHWAN

Georgia Institute of Technology

To meet end users' quality-of-service (QoS) requirements, online quality management for multimedia applications must include appropriate allocation of the underlying computing platform's resources. Previous work has developed novel operating system (OS) functionality for dynamic QoS management, including multimedia or real-time CPU schedulers and OS extensions for online performance monitoring and for adaptations, as well as QoS-aware applications that adapt their behavior to gain additional benefits from such functionality. This article describes a general OS mechanism that may be used to implement a wide variety of online quality management functions. ECalls is a communication mechanism that implements multiple cross-domain calling conventions that can be customized to the quality management needs of applications. The ECalls mechanism is based on the notions of events, event channels, and event handlers. Using events, applications can share relevant QoS attributes with OS services, and OS-level resource management services can efficiently provide monitoring data to target applications or application managers. Dynamically generated event handlers can be used to customize event delivery to meet diverse application needs, for example, to achieve high scalability for Web servers or small jitter for real-time data delivery.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Scheduling*; D.4.4 [**Operating Systems**]: Communications Management; D.4.7 [**Operating Systems**]: Organization and Design; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems

General Terms: Design, Performance

Additional Key Words and Phrases: Event delivery, operating system, quality-of-service, real-time events, quality management, dynamic code generation

1. ECALLS—A MECHANISM FOR ONLINE SYSTEM MANAGEMENT

1.1 Background and Motivation

Lack of quality-of-service (QoS) support from operating systems can be a detriment to the efficient implementation of applications like distributed virtual environments, multiplayer games, telepresence, remote sensing, or remote collaboration. This is because these applications' multiple and continuous data streams require bounds on latency, data loss, and jitter, to prevent audible gaps in audio or choppy

Authors' addresses: C. Poellabauer, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556; email: cpoellab@cse.nd.edu; K. Schwan, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332; email: schwan@cc.gatech.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2005 ACM 1551-6857/05/0800-0001 \$5.00

replay of video, for example. Previous work has addressed these needs by enhancing operating systems with resource management techniques for multimedia and real-time applications, where the applications and the resources being managed run in different *protection domains* and interact via well-defined operating system (OS) interfaces. There are three well-known problems with these approaches:

- Cross-domain calls like signals can be expensive, which can result in high delays in the transfer of control and data between caller and callee. Such delays are detrimental to service responsiveness and limit the scalability of applications with high call frequencies (e.g., Web servers).
- Cross-domain calls can be restrictive, in that they may not implement the functionality needed by QoS-aware or real-time applications.
- Multimedia and other real-time applications not only require timely communication between remote hosts, but in order to attain suitable end-to-end delays, they also require timely and *integrated* actions from multiple components within a host. General-purpose operating system components like CPU schedulers, however, are not built to respond to specific events of importance to applications, such as the arrival of data at a network interface, thereby causing undue delays in event response.

1.2 ECalls—A Uniform Mechanism for Cross-Domain Transfers

In comparison to previous work focused on specific applications or services, *ECalls* provides a single, uniform cross-domain transfer facility for control and data that is (1) sufficiently flexible to support a wide range of QoS-aware applications or services and (2) customizable to specific application needs. ECalls permits QoS-aware applications to communicate with OS-based resource management services to monitor resource availability, to renegotiate QoS specifications, or to be notified when an application should change its behavior (e.g., adapt its image quality to reduce processing or networking requirements [Poellabauer et al. 2002]). ECalls provides multiple methods for implementing such cross-domain events. For instance, consider the notification of single-threaded Web servers about the arrival of new service requests, which is commonly done in a pull-based fashion using *select()* or *poll()* system calls. ECalls can be used to implement alternative pull methods, thereby addressing the known poor scalability of both *select()* and *poll()* with high request frequencies [Chandra and Mosberger 2001]. A key component of ECalls is its support for better integrating the actions of the multiple software components involved in delivering some end user service, by linking event delivery with CPU scheduling. In this fashion, applications are not only notified of relevant events like network-level data arrival, but they can also be scheduled to respond to such events so as to meet certain end-to-end service requirements.

ECalls is a flexible cross-domain communication mechanism that uses *event channels* to allow parties interested in certain types of events to subscribe to shared channels to which events are submitted or from which they are received. Event submission and receipt result in actions being triggered, using *handler* functions associated with event channels. Handlers are defined at the time of subscription to the event channel, enabling online data transformation, event filtering, and similarly useful functionality. Both event formats and the control semantics of event channels are defined at the time of channel creation. Concerning semantics, there are multiple well-defined, per-channel ways in which control is passed between applications and the system domain upon event production and receipt. ECalls offers different mechanisms to transfer both data and control between different protection domains, thereby addressing the needs of a variety of applications and usage scenarios. For instance, a “real-time” event channel implies that, upon event generation by the kernel, a real-time signal will be generated to the address space that is subscribed to this channel.

The notion of ECalls and their use for QoS management is based on past work in system monitoring and in system control or adaptation [Bihari and Schwan 1991], where event-based paradigms are used to represent and manage monitoring data and to control system operation [Pietzuch and Bacon

2002]. Another basis for ECalls is the event-based communication in Internet-wide applications, which has received increased attention in part because of its support for decoupled communication: event producers are unaware of number or identities of event consumers (i.e., anonymous communication), and events can be raised at any time without the producer waiting for a response from the consumer and without the consumer having control over when events are raised (i.e., asynchronous communication). In such environments, event delivery typically implies *cross-domain control transfers*, where the event-related action (e.g., the execution of a handler routine) is performed in the context of the consumer process. This can require a *processor switch* to the consumer if it is not a currently active process. Alternatively, the event-related action can be performed by the event producer (e.g., an operating system service), where the action is performed by the producer on behalf of the consumer without the need for a control transfer. Events are accompanied by data, as part of the event itself and explicitly described by its data format or as separate data items described as external attributes. When event delivery includes a cross-domain control transfer, data delivery results in *cross-domain data transfer*, which consists of copying the data associated with the event to a memory area accessible by both the event producer and consumer.

The implementation of ECalls as a dynamically loadable kernel module leverages the fact that it has become increasingly easy to extend operating systems with new functionality. In Unix-based systems, extensibility is used to realize complex and demanding services, like load-balancing mechanisms in parallel computing environments, facilities that support distributed resource management and QoS for real-time applications, or kernel ports of user-level application components like the Linux in-kernel HTTP Web accelerators *tux*¹ and *khttpd*.²

1.3 Contributions

ECalls is a mechanism for online system management, supporting both the QoS management commonly required in real-time and multimedia systems and the self-management that has become increasingly important for complex distributed systems and applications [Birman et al. 2003]. The idea is to provide a general mechanism for linking kernel- with application-level actions and knowledge in distributed systems:

- ECalls provides multiple cross-domain event call methods (e.g., shared memory, signals, kernel-level handlers) that can be used and combined to implement the interactions necessary for providing the QoS required by applications.
- Event filters* can initiate lightweight cross-domain control transfers and attach constraints to such transfers before other, more heavyweight control transfer facilities are utilized. This can be used to preprocess data associated with events or to provide behavior analogous to that of optimistic active messages [Wallach et al. 1995].
- Custom event handlers* can be deployed—or even dynamically generated—by applications, in order to specialize the system’s behavior to the needs of the application using it.
- Event-aware CPU scheduling* links an operating system’s CPU scheduler with the event delivery mechanisms of ECalls, such that processes with pending events are given preference over other processes, thereby increasing real-time and multimedia applications’ responsiveness to these events.

The results presented in this article underline the benefits of ECalls’ flexible and low-overhead event-based communications between protection domains. For a video streaming application, ECalls succeeds in providing end users of video players with the quality they expect, avoiding high end-to-end delays

¹<http://www.redhat.com/products/software/tux>.

²<http://www.fenrus.demon.nl>.

and large jitter by coordinating event delivery (where each event triggers the display of a new video frame) with process scheduling. In the case of a Web server, ECalls events are used to improve server behavior in overload situations, by increasing their reply rates. In the experiments presented in this article, we demonstrate that the use of ECalls as a replacement for existing well-known communication approaches (e.g., the *select()* system call) between OS and applications results in an increase in reply rates of about 50%.

1.4 Structure of the Article

In the following section, ECalls is related to other past and ongoing research efforts in multimedia and operating systems. In Section 3, we discuss the implementation details of ECalls, including its application programming interfaces (APIs). This is continued in Section 4, which focuses on ECalls' ability to coordinate CPU scheduling with event delivery. Section 5 presents experimental evaluations and Section 6 concludes the article with a summary of the results.

2. RELATED WORK

Multimedia and real-time applications require resource management support from the underlying operating system to achieve their real-time and QoS guarantees. This has led to the development of new operating system services for various resource management tasks, for example, focusing on process scheduling [Nieh and Lam 2003]. Since an application's interactions with such kernel-level services can be costly, researchers have sought ways to control call overheads [Druschel and Peterson 1993], and they have attempted to reduce the frequency of system calls, by extending kernels with appropriate application-specific functionality [Bershad et al. 1995; Engler et al. 1995]. In addition, upcall primitives have been introduced [Clark 1985], to better integrate the kernel- and application-level actions carried out for certain requests. Real-time variants of upcalls address the specific needs of multimedia and real-time applications [Gopalakrishnan and Parulkar 1998]. ECalls can implement all of the upcall methods developed in such research.

Common elements of the solutions mentioned above are (1) the need to share information about performance-critical events between kernel- and user-level facilities, and (2) to be able to act on such information in a timely fashion. For instance, communication rates can be adjusted based on information about buffer fill-levels [Steere et al. 1999], if such information is made available and acted upon with little delay. In fact, past work has shown that system quality may be reduced rather than improved by runtime adaptation if such actions are not performed within certain tolerances [Rosu et al. 1997]. By linking CPU scheduling with events, ECalls can provide applications with the timeliness guarantees they need.

A particular direction of research has addressed the poor scalability of system calls like *select()* or *poll()*. In Druschel and Peterson [1993], the authors implement an integrated buffer management and transfer mechanism optimized for high-bandwidth I/O. The goal is to achieve high throughput across protection domains by exploiting page remapping and shared memory techniques. Yau and Lam [1996] introduced an approach that efficiently transfers data and control between application and system domain and also provides rate-based flow control. This is achieved by using I/O-efficient buffers and independently scheduled kernel threads. Banga et al. [1999] introduced an event delivery system that allows applications to register interest in event sources like sockets and collect these events at a later time. Similar implementations include the ones addressed in Lemon [2001], Rosu and Rosu [2003], and Pai et al. [1999b]. Typically, these event delivery mechanisms are pull-based, where applications have to scan some form of lists, flags, or queues, whereas ECalls is able to notify a process of pending events by executing a handler function on its behalf and/or raising its scheduling priority. The result is

an event-aware scheduling strategy that not only considers an application's scheduling attributes, but also the events waiting for delivery to this application. For example, a video player waiting to display the next image can be given preference over other applications if the next image has been received via the network interface and is ready to be displayed. This results in smaller jitter and smoother video replay. But ECalls also supports the push-based delivery of events, as exemplified by real-time signals or by direct invocations of handler code—including dynamically generated code—in the system domain.

ECalls generalizes the upcall methods described in Gopalakrishnan and Parulkar [1998] by offering a variety of communication techniques and by linking CPU schedulers with event delivery. Pai et al. [1999a] introduced a new flexible and general I/O approach that avoids data copying. I/O completion ports, supported in Windows NT, use a number of preforked threads to handle incoming events. A throttling mechanism limits the number of currently active threads to avoid large context switching overheads.

The idea of using events to share knowledge across protection and subsystem boundaries builds on earlier work reported in Manimaran et al. [1998], for example, where the authors proposed an integrated framework that permits process and message schedulers to interact in distributed real-time systems. Similarly, in Lee et al. [1996], an architecture that is aware of the real-time characteristics of tasks sending and receiving network packets was introduced. The goal was to overcome the traditional deficiencies like FIFO ordering of incoming packets and processing in the kernel of all packets regardless of their priority to the receiving application.

3. ECALLS—DESIGN AND IMPLEMENTATION

We next outline the design and implementation of ECalls. The intent is to provide sufficient detail to motivate the use of ECalls for online quality management, as explained in the subsequent sections. The key components of ECalls are

- shared memory between applications and kernel services for both event notification and data sharing (two separate memory areas for communication from application level to kernel level and vice versa);
- multiple event notification approaches, such as signals and kernel event handlers;
- dynamic generation of kernel-level event handlers; and
- coordination of event delivery and CPU scheduling.

ECalls has been implemented as a dynamically loadable kernel module for Linux 2.2.13 and 2.4.19. It uses event channels to link event producers (e.g., kernel services) and event consumers (e.g., applications). As a result, kernel services like device drivers, load balancers, or resource managers can raise events that are passed on to one or more applications via ECalls. If there are multiple consumers of an event, the order of event notification depends on the applications' CPU scheduling priorities if a processor switch is required. Otherwise, their event handlers are simply invoked in the order in which applications register them with the kernel service. ECalls offers several registration interfaces for applications and kernel services:

- Service Registration Interface*. Kernel services announce their presence by registering with this interface, specifying unique names to identify themselves. These names are exported via the `/proc` virtual file system, where applications can obtain a list of all currently available services.
- Event Channel Subscription Interface*. Applications express their interests in the events published by services by registering through this interface. An event channel (and therefore a kernel service) is identified by the name of the service found in `/proc`.

—*Kernel Handler Registration Interface*. Kernel services and kernel modules can use this interface to register kernel-level functions, which can be executed by ECalls on behalf of applications (as kernel event handlers).

The registration of a new kernel service has the following syntax:

```
kernel_service {
    ...
    service_id = register_service(NAME_OF_SERVICE);
    while (service_needed) {
        /* perform kernel service (e.g., resource management) */
        ...
        /* raise event */
        raise_event(service_id, pid, data_pointer, deadline, cpu);
        ....
    }
    unregister_service(NAME_OF_SERVICE);
}
```

Event notification is performed by invoking the *raise_event* function with the following attributes:

- service_id*. This attribute contains the unique identifier returned by the service registration interface and is used by ECalls to associate an event producer (kernel service) with event consumers (applications).
- pid*. Even if multiple applications subscribe to the same event channel, a service is able to direct an event to only one event subscriber (identified by the process ID). If all event subscribers are to receive the event, *pid* is -1 and ECalls notifies all processes registered for this event. A *wake-one* policy as supported with the *pid* attribute is desirable if, for example, multiple server threads wait for requests on a socket. The kernel can then notify only one of these servers instead of all of them (e.g., in a round-robin fashion).
- data_pointer*. This attribute of type *unsigned long* can be used to pass data along with an event, either as a simple unsigned long value or it can be pointer to a kernel- or user-level memory location holding the data to be shared.
- deadline*. Events can have an associated deadline (expressed in microseconds); if present, events are dispatched to applications according to the EDF scheduling policy and events that miss their deadline are discarded from the event queue.
- cpu*. This attribute is a flag indicating if the kernel service wishes to take advantage of ECalls' ability to cooperate with the CPU scheduler.

An application registers for events as follows:

```
main {
    struct ecalls my_ecalls;
    struct sh_memory my_memory_up;
    struct sh_memory my_memory_down;
    ...
    my_ecalls.signal = SIGRTMIN + 0;
    my_ecalls.process_id = getpid();
    my_ecalls.memory_up = &my_memory_up;
```

```

my_ecalls.memory_down = &my_memory_down;
ECalls_subscribe(NAME_OF_SERVICE, &my_ecalls);
...
while (running) {
    ...
}
ECalls_unsubscribe(NAME_OF_SERVICE);
}

```

To specify how an application wishes to be notified, an application initializes a data structure (*struct ecalls*), which has the following entries:

- signal*. This specifies the real-time signal number, where it is the responsibility of the application to implement a signal handler for the chosen signal number.
- process_id*. The application can specify whether the event should be received by itself or by some other process, in both cases identified by the process ID.
- k_handler*. This string identifies the name of a kernel event handler that has been previously registered with ECalls and will be called by ECalls on behalf of the application when an event is raised.
- k_code*. This string identifies C-like code that will be “downloaded” into the kernel by ECalls, compiled, and added to the list of available kernel event handlers.
- k_thread*. This flag indicates whether a kernel event handler is to be executed in the context of a kernel thread or at interrupt context.
- memory_up* and *memory_down*. These are pointers to the two shared memory segments.

With ECalls, an application can register two shared memory segments, one for data transfer from user to kernel level and one for the opposite direction. These memory segments can be used for event notification, for example, a kernel service can toggle a memory-resident flag, which is being polled periodically by the application. Further information passed in *struct ecalls* includes the order of event notification (e.g., first *k_handler*, then *signal*) if more than one method is desired. After the data structure is initialized, the application registers interest in a kernel service and the associated event channel by calling *ECalls_subscribe*, specifying the name of the kernel service and the above-mentioned data structure.

3.1 Event Notification

A kernel service (e.g., QoS manager, load balancer) raises an event and associates a *deadline* with the event. The event is added to an event queue that is ordered earliest-deadline-first. Events are taken from the head of the queue and delivered to all consumers (see Figure 1). Each consumer has one or more handler actions associated with an event, for example, the execution of a kernel event handler or the raising of a real-time signal. If more than one action has been registered, the actions are performed one-by-one, and an action’s “result” (e.g., return value) determines whether a subsequent step is skipped. The remainder of this section describes the different control transfer methods supported by ECalls.

3.1.1 Real-Time Signals. The POSIX.4 standard extends the signal interface with real-time signals. Unlike regular signals, real-time signals are queued and can carry a small amount of data. Chandra and Mosberger [2001] showed that real-time signals are an efficient mechanism, providing good throughput compared to *select()* or *poll()* system calls. To ensure predictability and high responsiveness in the dispatching of real-time system calls, we modify the signal-handling sequence as implemented in Linux 2.4.19. In particular, Linux supports 32 regular and 32 real-time signals. The signals are identified by

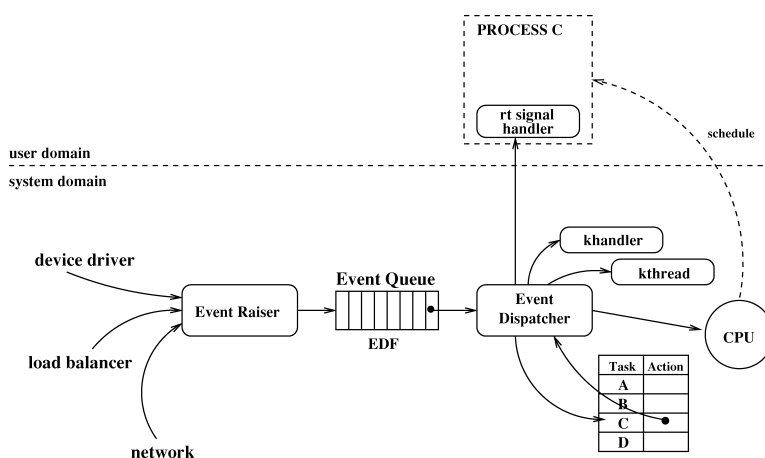


Fig. 1. ECalls event delivery architecture.

a 64-bit variable, each bit indicating if a signal has been raised or not. In the original implementation, the lower 32 bits are checked first, and if a bit is set, the corresponding signal handler is invoked. However, the lower 32 bits correspond to the regular signals, so that regular signals are handled before real-time signals. Keeping in mind that regular signals can be “caught” and handled by user-specified signal handler code (except SIGKILL and SIGSTOP), this can lead to delays to the invocation of—potentially more important or time-constrained—real-time signals. Therefore, we change the sequence to the following order: (a) SIGKILL and SIGSTOP are checked first (the only two signals which can not be caught by the user and which always result in termination or stopping of the process), (b) all real-time signals are checked, with SIGRTMIN having the highest priority and SIGRTMAX having the lowest priority of all real-time signals, and (c) all remaining regular signals are checked and handled if required.

3.1.2 Kernel Event Handlers. Kernel event handlers can either be provided by kernel-loadable modules or they can be dynamically inserted into the kernel by applications. The main advantage of using kernel event handlers on behalf of applications is the small overhead of handler invocation, since no “upcalls” are needed. Kernel event handlers can also be run in interrupt contexts, making context switches unnecessary, which further contribute to the overheads of user-level solutions. An example of the use of such an event handler leverages the recent trend of implementing HTTP accelerators in kernel space. A kernel event handler can be invoked through activity on a socket and, as a consequence, the handler reads the request from the socket, analyzes it, and decides whether to service the request directly from within the kernel (e.g., static Web requests) or whether to pass the request on to an application Web server (e.g., dynamic Web requests).

3.1.3 Kernel Threads. Kernel event handlers are a powerful and efficient way to handle events. However, if an event is dispatched outside of a process context (e.g., during a timer interrupt) and the event handler would run too long if executed at interrupt context, the event handler can be invoked within the context of a kernel thread provided by ECalls. ECalls maintains a pool of preforked threads, where the pool size is dynamically modified as needed.

3.1.4 Shared Memory. The memory areas shared between an application and a kernel service can be used to notify an application of kernel-level events. Here, instead of executing costly system calls to obtain information about kernel-level events, an application can simply scan entries in the shared memory, which are modified by the kernel service whenever an event is raised.

3.1.5 *Execution Context.* If an application is notified of an event through real-time signals or via shared memory, the event is handled in the context of the application. However, when a kernel thread is executed, the event is handled in the context of that kernel thread. If this kernel thread has to access resources of the application, such as file and socket descriptors, certain provisions may be necessary to overcome access restrictions. For example, as part of the ECalls mechanism, we modify the kernel source such that a kernel service is able to access the file descriptors of the applications that register with this kernel service via ECalls. That is, kernel handlers can implement parts of application-level functionality, for example, to preprocess incoming data on socket connections. Further, if kernel event handlers are used, the handler function may be executed in interrupt context. Again, provisions have to be made to ensure that an application's resources can be accessed. The details of these kernel modifications are beyond the scope of this article.

3.2 Cross-Domain Data Transfer

The shared memory areas in ECalls are used for event notification and for the exchange of data between application and OS extensions. Using two separate memory areas minimizes the need for synchronization between application and kernel service. The memory areas' structures are described in a C header file, organized in one of two possible ways. In either case, the first entry is an integer value (called *flag*), which can be modified each time an event is generated or handled. The following code shows the two possible memory structures:

```

struct sh_memory {
    int flag;
    unsigned long bit_pattern[MAX];
    [data part]
};

struct sh_memory {
    int flag;
    int front;
    int back;
    [data part]
};

```

In the first case, an array called *bit_pattern* holds a bit per data entry in the following data part. When an event is generated, *flag* is incremented, the event data is written into the corresponding position in the data part, and the corresponding bit in *bit_pattern* is set. In the second case, the memory segment can be structured as a ring buffer. Then, the *flag* entry is followed by a *front* and a *back* entry, pointing to the beginning and the end of the momentarily used part of the memory segment, respectively. While the goal of the shared memory is to facilitate data sharing between the application and the kernel domains, other domains can also utilize this feature, for example, network processors can share data efficiently with applications or storage devices, thereby avoiding costly data copies.

3.3 Dynamic Handler Generation

An important attribute of ECalls is its ability to support *dynamic instrumentation* of kernel functionality, by allowing applications to insert dynamically generated code into a running kernel. Unlike kernel modules, this feature supports simple functions that can be shipped between systems as strings and translated into native machine code by a lightweight in-kernel compilation component. These functions are expressed in *E-code*, a C-like language that has been developed as part of the ECho event service [Eisenhauer et al. 2001]. E-code is inspired by Icode, an internal interface developed at MIT as part of the 'C project [Poletto et al. 1996]. For ECalls, the Georgia Tech E-code code generator has been ported to the Linux kernel and consists of two loadable modules. Currently, E-code supports the C operators, *for* loops, *if* statements, and *return* statements. While these limitations restrict the capabilities of E-code, they also facilitate the protection from malicious code. However, work is in progress to extend E-code's capabilities (e.g., adding dynamic memory management and

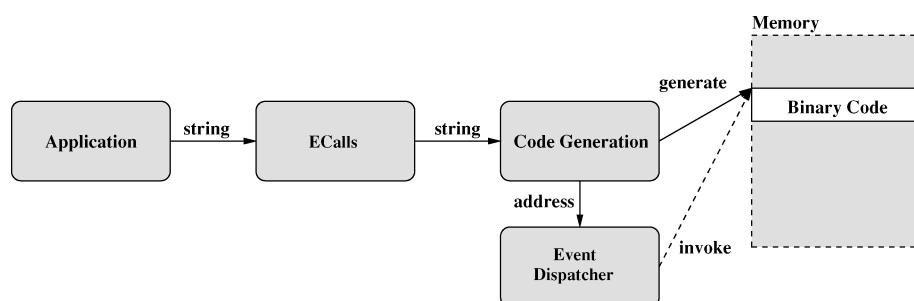


Fig. 2. Dynamic code generation.

pointers), while at the same time adding protection mechanisms. In addition, other efforts [Ganev et al. 2004] have contributed hardware-based protection mechanisms that can be used in conjunction with ECalls.

In order to generate and install new code in the kernel, an application passes a string carrying the code (e.g., with a system call or via `/proc`) to the ECalls mechanism, where the code is translated into machine code. When the corresponding kernel service raises an event, ECalls invokes the newly generated kernel event handler on behalf of the application. Figure 2 shows how code is deployed in ECalls. An application passes the code as string to ECalls, where it is passed to its “dynamic code generation” component. The string is parsed and translated into binary code and placed into memory. The location of the newly generated binary code is shared with the event dispatcher, which will invoke the new code whenever an event for the corresponding application occurs. The following is a simple example of an E-code function, where resource attributes—possibly collected by a resource monitor—are inspected. With E-code, parameters can be passed as basic types (e.g., integer, char) or as structures as shown with *input* in the sample code. Similarly, return results can be basic types or structures (e.g., *output* in the sample code).

```

char * my_code = ‘‘{
    int i;
    int j;
    for (i = 0; i < NUM_RESOURCES; i++) {
        for (j = 0; j < input.resource[i].num_attributes; j++) {
            if (input.resource[i].attribute[j] > input.resource[i].threshold[j])
                output.resource[i].exceeded[j] = 1;
            else
                output.resource[i].exceeded[j] = 0;
        }
    }
}’’

```

4. EVENT-AWARE CPU SCHEDULING

The timely delivery and processing of events is particularly important for time-constrained applications such as multimedia streaming, virtual environments, or interactive distributed simulations. Consider, for instance, a distributed game for which game events like position updates must be delivered in a timely fashion. Here, by coordinating task scheduling with important game events (such as player movements or shots in a first-person shooter game), it is possible to increase the responsiveness to these player actions. The quality of video replay is determined by the experienced end-to-end delays

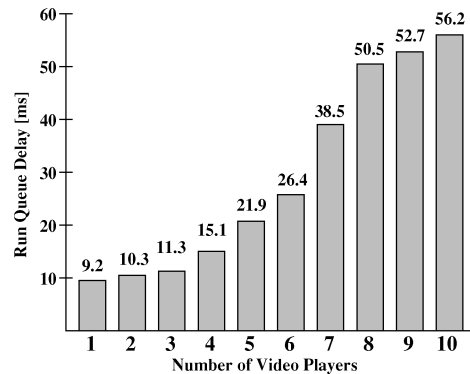


Fig. 3. Average run queue delays for a number of video players that have to compete with each other and with another real-time process (running in an endless loop) for the CPU.

and jitter. Here, techniques like proportional share scheduling of tasks and communications can reduce jitter for continuous media streams. However, an event-aware CPU scheduling approach can ensure that video players are scheduled exactly when a new video frame has to be displayed, minimizing the jitter experienced by the user. We next illustrate the opportunities derived from using ECalls for purposes like these, using a distributed video player as the driving application.

Typical distributed video player implementations use *timed waits* to achieve the *interframe times* necessary for their desired frame rates. In other words, such an application *sleeps* for a certain amount of time, and when it *wakes up*, it is placed back into the run queue of the CPU scheduler. However, the delay between the point when this application becomes schedulable (i.e., wakes up) and when it begins to run (i.e., enters the “running state”), termed *run queue delay*, varies depending on the scheduling policy implemented, the scheduling attributes assigned to this and other schedulable applications, and the current CPU load. Run queue delays can increase latencies and jitters for continuous media streams, and they can reduce the responsiveness of real-time applications like distributed games. For instance, when running on a general-purpose operating system like Linux, a single video player can experience significant run queue delays when it has to compete with a second real-time process due to the coarse granularity of the system’s time base, which is 10 ms on Intel-based Linux systems. When it has to compete with other video players for the same CPU, run queue delays increase substantially, resulting in significant variations in interframe times even for a small number of video players, as shown in Figure 3. This graph shows the average run queue delays for all video players, measured on a Pentium II processor with 300 MHz, 256 MB RAM, running Linux 2.2.13. The scheduling policy for all tasks is the round-robin real-time scheduler offered by Linux. Using ECalls, the arrival of a message at a network connection triggers an event, the ECalls mechanism then not only notifies the application of the arrival of the event, but it also cooperates with the CPU scheduler to ensure the timely delivery of the event. The result is that the CPU scheduler is made “aware” of important application events. Therefore, ECalls offers the basic functionality needed for creating “event-aware” systems, by linking the scheduling of processes with the delivery of events for these processes. The effect is that processes acting as sinks of events are favored over other processes whenever they receive events. This represents one example of how ECalls may be used to implement quality management policies in which applications coordinate with certain system services.

The approach presented here is based on the Dynamic Window-Constrained Scheduler (DWCS) real-time scheduler [West and Poellabauer 2000]. Note that the implementations of the event and CPU schedulers are separate, thus permitting the event scheduler to be linked with any CPU scheduler

and facilitating the porting of ECalls to systems using other CPU schedulers. As a result, ECalls' event scheduler *revises* the CPU scheduler's decisions as shown in this section. The event scheduler discussed in this article depends strongly on the policies implemented by the CPU scheduler, in order to maximize the responsiveness of applications to events. Other CPU schedulers will require different policies for the event scheduler, however, "generic" event scheduler implementations are also possible, which will conservatively revise any CPU scheduler's decisions. However, the generality of such an approach would limit the increase in responsiveness achievable.

4.1 DWCS CPU Scheduling

The traditional UNIX scheduler has been shown to have unacceptable performance for multimedia applications [Nieh et al. 1993]. For example, an application with a fixed real-time priority could have precedence over all other applications at all times, and therefore, starve best-effort applications. This has led to the development of new scheduling approaches, including those based on reservations and on proportional share resource allocations. To efficiently support real-time applications, we use a hard real-time CPU scheduler mentioned above, called *Dynamic Window-Constrained Scheduler* (DWCS) [West and Poellabauer 2000]. DWCS assigns each process the following attributes: a period T , a service time C , and a window-constraint x/y . Using these attributes, DWCS *attempts* to service a process for at least C time units in a period of T time units, and it *guarantees* that it will service a process in $y - x$ periods in a window of y periods if the *CPU utilization* is less than or equal to 100%. The period T_i of a process i is used to set a deadline until the scheduler has to service process i for at least C_i time units. If the process misses its deadline more than x_i times in a window of $T_i * y_i$, the scheduler *violated* the real-time guarantees to this process. Each process can be scheduled once in its period, unless it is marked as *work-conserving*, in that case it is possible to schedule this process several times within its period as long as CPU utilization allows.

Scheduling attributes are adjusted dynamically to reflect the progress of a process. We distinguish between the *original window-constraint* x/y and the *current window-constraint* x'/y' , where the latter is modified dynamically (details can be found in West and Poellabauer [2000]). The precedence rules used by DWCS among processes are as follows: (1) order processes earliest deadline first (EDF); (2) if deadlines are equal, order processes with tightest window-constraints first; (3) if deadlines are equal and the window-constraints are zero, order highest window-denominator first; (4) if deadlines are equal and the window-constraints are equal (and nonzero), order lowest window-numerator first; (5) in all other cases: first-come first-serve.

The remainder of this section describes the cooperation between the ECalls event scheduler and DWCS, where the goal is to maximize event responsiveness without compromising the real-time guarantees provided by DWCS.

4.2 Event Scheduling with DWCS

Cooperation between ECalls' event scheduler and the CPU scheduler essentially ensures preferential CPU scheduling for processes for which ECalls events are available. Specifically, whenever the ECalls' event queue is nonempty, the event scheduler is invoked each time the CPU scheduler runs. After the CPU scheduler finishes selecting the next process, the event scheduler compares the scheduling attributes of the next process with the attributes of the sink process for the first event in the event queue. If the event scheduler determines that the sink process can be scheduled *before* the process chosen by the CPU scheduler, without violating the real-time guarantees given to the process chosen by the CPU scheduler, the sink process is scheduled instead.

Assume that process i is the process selected by DWCS and process j is the sink of the first event on the event queue. The event scheduler applies the following five rules to processes i and j :

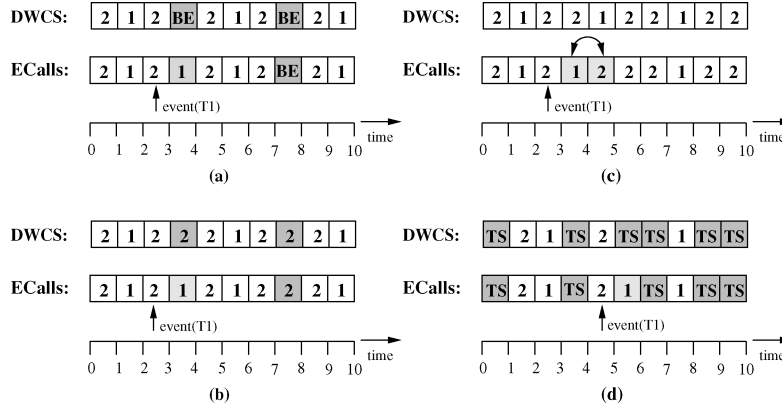


Fig. 4. Examples for (a) Rule 2, (b) Rule 3, (c) Rule 4, and (d) Rule 5.

- Rule 1.* If $j = i$ (i.e., DWCS selected the sink process), the only action the event scheduler has to perform, is to remove the event from the event queue.
- Rule 2.* If task i is a best-effort task, ECalls replaces i by j and removes the event for process j from the event queue. DWCS schedules best-effort processes only if all runnable real-time processes have been serviced within their respective periods and none of them is a work-conserving process. That means further that process j receives an additional time unit in its current period, so that it is able to react to an event immediately. No real-time guarantees are compromised since all real-time processes have been serviced in their corresponding periods (Figure 4(a)). Note that a certain amount of CPU utilization (e.g., 5%) can be reserved for best-effort tasks, preventing the starvation of these tasks.
- Rule 3.* If process i is a work-conserving process that received at least C_i time units of CPU time in its current period T_i , the event scheduler replaces i with j and removes the event for process j from the event queue. The real-time guarantees of i are not compromised in this case, since process i received C_i time units in its current period already (Figure 4(b)).
- Rule 4.* Assume that both processes i and j have not been serviced in their current periods yet, and both have the same deadline. Further assume, that DWCS selected process i as the next running process due to its tighter window-constraint compared to process j . ECalls' event scheduler gives process j preference over process i , if this does not lead to a missed deadline for i (i.e., $\Delta t - C_j - C_i > 0$, where Δt is the remaining time in period T_i). In other words, process i will be delayed by C_j , but since its deadline will not expire, DWCS will select this process after process j has exhausted its service time C_j (Figure 4(c)).
- Rule 5.* In addition to the rules above, we introduce the notion of a *task server*, which is a pseudoprocess with scheduling attributes determined as follows:

$$x_{ts}/y_{ts} = 0/y_{\max}, y_{\max} = \max\{y_i\} + 1.$$

This assigns the task server the tightest window constraint possible. The service time C_{ts} is the same as the service time of the sink process of the first event in the event queue, or 1 otherwise. The *rest utilization* U_r of the system, which is the *maximum utilization* minus the *current utilization*, is used to determine the value of the period T_{ts} :

$$T_{ts} = C_{ts}/U_r.$$

The attributes for the task server have to be re-calculated when the service time of the first event in the event queue changes (e.g., when the first event has been delivered and the new event at the front of the queue has a different service time). Each time the task server is selected by DWCS, the event scheduler replaces it with the sink of the first event in the event queue (Figure 4(d)). If there are no events pending, a best-effort task can be scheduled instead. The purpose of the task server is to *reserve* the remaining CPU time for processes that have events pending.

Figure 4 summarizes these rules, where “BE” indicates a best-effort task, and TS indicates that the scheduler selected the task server. In each graph, the top part shows the scheduling output of DWCS, while the bottom part shows the scheduling output revised by ECalls. In graphs (a) and (b), task T_1 has the following attributes: $T = 4$, $C = 1$, $x/y = 1/2$; task T_2 has the following attributes: $T = 2$, $C = 1$, $x/y = 1/4$. In both cases, T_1 is being notified of an event at time 2.5; however, in (a) both tasks are non-work-conserving, while in (b) they are both work-conserving. In graphs (c) and (d), both tasks have a period $T = 3$ and a service time $C = 1$. T_1 's value for x/y is $1/2$, while T_2 's value for x/y is $1/4$. Again, both tasks are work-conserving. In (c), the event is raised at time 2.5, while in (d) the event is raised at time 4.5. Further, in (d), the task server's period is computed as follows: $T = C/U_r = 1/0.58 = 1.7 \Rightarrow T = 2$.

4.3 Summary

Applications like multimedia, games, virtual environments, or scientific visualization require the timely delivery and processing of events relevant to their operation. These events can be system-level activities like exceptions, the arrival of data at a network connection, or the announcement of the availability of a resource for which an application has been waiting. This section introduced an approach to coordinating the task scheduling of an operating system with the dispatching of events of the ECalls mechanism. The outcome is that it is possible to increase the responsiveness of quality-aware applications to events, for example, to improve end-to-end delays or jitter experienced by communicating processes. While the implementation of ECalls' event scheduler is kept separate from the CPU scheduler (thereby avoiding any modifications to the actual scheduling code), in order to achieve the best possible responsiveness, the rules applied by the event scheduler depend strongly on the policies implemented by the CPU scheduler.

5. CASE STUDIES AND EXPERIMENTAL EVALUATION

This section describes the use and experimental evaluation of ECalls-based systems. The first part discusses some microbenchmarks, followed by an introduction of a kernel extension that raises ECalls events whenever network activity is monitored. These events are then passed to the appropriate application, as shown with the Web server or video player used in this section. We then evaluate the performance improvements achievable with ECalls, expressed in increased reply rates for a Web server, or the improved user-perceived QoS for a video player application.

5.1 Microbenchmarks

The following numbers have been obtained on an AMD Athlon computer with 550 MHz, 64-MB RAM, running Linux 2.2.13. Here, real-time signals require 26 μ s; however, if the process receiving the signal is not currently active, the response time is much larger because the process has to be scheduled first. In contrast, if ECalls invokes a kernel-level handler function, the overhead is only 2.1 μ s. To add an event to the event queue in ECalls requires 200 ns and the overhead caused by ECalls' event scheduler is about 1.2 μ s if the event queue is nonempty.

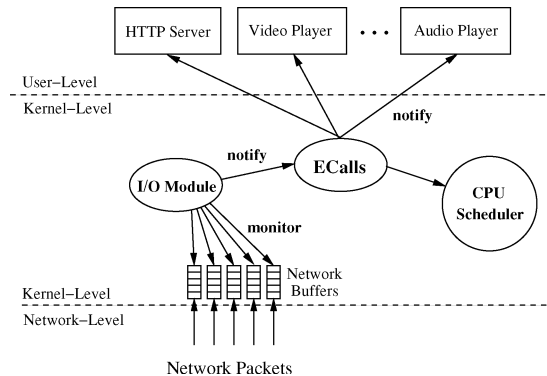


Fig. 5. Notification of socket activity with an I/O event delivery module and ECalls.

5.2 Implementation of an I/O Event Delivery Module

Unix systems provide *select()* and *poll()* system calls, which query a set of file descriptors passed in an array for activity. The system call returns when there is activity in at least one of these descriptors or when the system call times out. The application then has to scan a returned array to find the descriptors that are actually active. Unfortunately, Web servers like Zeus, Flash, or *thttpd* are known not to scale well to thousands of file descriptors because of their use of the *select()* approach. The problem is that the kernel has to scan the entire array of socket connections each time a system call is executed.

ECalls may be used to implement a scalable I/O event delivery module, henceforth called the *I/O module*. Applications register their interests in sockets via the ECalls mechanism. If data arrives at one of these sockets, the registered application is notified using one or more of the methods described earlier. A similar example has been presented in Banga et al. [1999], which introduced a scalable event notification mechanism to replace the expensive *select()* system call. To be able to support this notification mechanism, we added approximately 20 lines of code to the networking code inside the Linux kernel and one additional entry into the *sock* structure, the latter being a flag that can be used by the socket owner to express interest in event notification if socket activity is monitored. When the I/O module detects activity on one of the monitored sockets (Figure 5), it generates an event for the application owning this socket. Each application has two data structures, which are both locked into memory and shared between the application and the event delivery mechanism. The first data structure has the following entries:

```
struct socket_interest {
    int flag;
    unsigned long fd_list[MAX];
    unsigned long updated_fd_list[MAX];
};
```

The first entry, called *flag*, is incremented each time the application submits a change in interest. The next entry, called *fd_list*, is an array used to indicate which file descriptors the application has registered for; each bit in *fd_list* corresponds to a file descriptor. The second array, called *updated_fd_list*, is used to indicate the changes in *fd_list* since the last time the registration module read from this data structure. Its purpose is to accelerate the registration process.

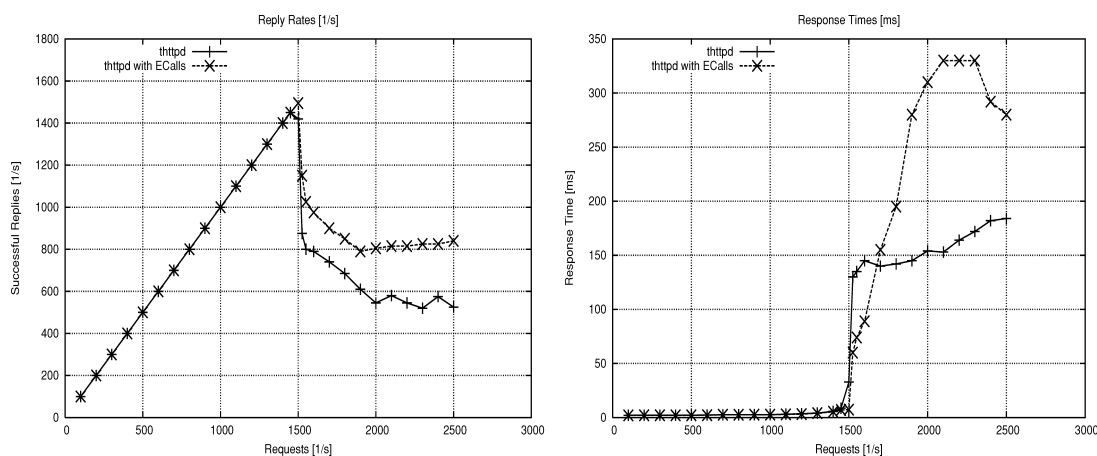


Fig. 6. Reply rates and response times for both the thttpd Web server and the modified thttpd Web server using ECalls and the I/O event delivery module.

The second data structure contains the following information:

```
struct socket_ready {
    int flag;
    unsigned long fd_active[MAX];
};
```

The value of *flag* indicates how many sockets are active, that is, how many sockets have data in the receive buffer. The actual file descriptors for the active sockets can be found in the next entry, called *fd_active*.

5.3 Experiments with a Web Server

This experiment modifies thttpd,³ which is a small, fast, single-threaded, event-driven Web server. Its performance is improved with the I/O event delivery module described above. Specifically, thttpd currently uses the *select()* system call for all HTTP requests. We modify its API such that thttpd subscribes every new incoming request with the I/O event delivery module. Then, instead of using a *select()* call, thttpd continues to service requests and selects the next connection to service via the *fd_active* array.

For experimentation, client requests are generated using the httpperf Version 0.8 performance tool. The HTTP server is a Pentium II with 450 MHz and 512-MB RAM, running our modified thttpd application. The client machines are five Sun Ultra 30 s with a 248 MHz processor and 128-MB RAM each. The machines are connected via a switched 100 Mbps Ethernet. In this experiment, the clients request a small static Web page for a duration of 180 s, each request with a timeout value of 1 s. Figure 6(a) shows the achieved reply rates with both the original thttpd server and our modified server using ECalls (thttpd-ECalls). The thttpd server is highly optimized, so that the difference in performance for small loads is not visible, as evident from the graph. On the other hand, for overload cases, thttpd displays poor behavior: its reply rate drops to 25% at request rates of 2000 per second. The reply rate for thttpd with ECalls also decreases in the case of overloads, but less dramatically (e.g., to 40% at a request rate of 2000 per second). Figure 6(b) shows the average response times for thttpd and thttpd-ECalls. Here, the response times of thttpd settle at approximately 150 ms when the server is overloaded, whereas

³<http://www.acme.com/software/thttpd>.

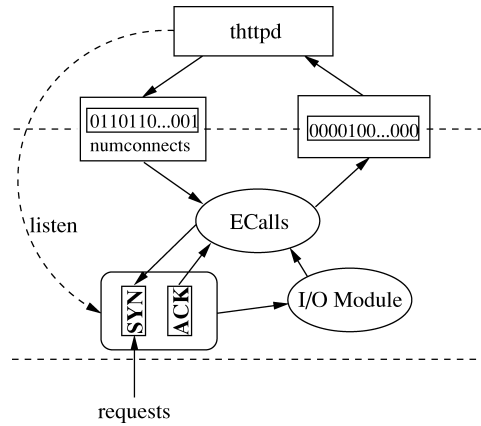


Fig. 7. Modified thttpd Web server using ECalls and the I/O module: ECalls monitors both the number of open connections (*numconnects*) and the buffer fill level of the listen queue with completed requests (ACK-queue) to determine the size of the listen queue with incomplete requests (SYN-queue).

in the case of thttpd-ECalls, the response time increases until it settles at approximately 300 ms. The reason for this behavior is the higher reply rate of thttpd-ECalls, where thttpd-ECalls is able to respond to more requests than thttpd, thereby resulting in higher response times. However, the response times in these graphs are still well below the timeout value of 1 s.

The next experiment investigates whether the use of ECalls can improve the overload behavior shown above. Provos et al. [Provos and Lever 2000; Provos et al. 2000] analyzed the overload behavior for the thttpd and phhttpd Web servers. In Provos and Lever [2000], real-time signals were used to notify the Web server of new requests and the number of signals, and therefore the number of pending requests is used as indicator for server overload. The overload behavior shown in Figure 6 is referred to as *receive livelock*; Mogul and Ramakrishnan [1997] suggested dropping requests as early as possible to achieve more request completions. While in Provos and Lever [2000] requests were dropped by the server if overload was detected, we use ECalls to drop requests early in the kernel. That is, we flood the server with requests, this time with a more complex Web page, and monitor overload behavior, but then we improve overload behavior by using the ability of ECalls to cheaply exchange information between user- and kernel-level. Specifically, the Web server continuously updates a new variable, *numconnects*, in the memory segment, telling ECalls the current number of open connections being serviced by the server. In addition, ECalls monitors the buffer fill level of the *completed connection queue* of the listening socket (ACK-queue in Figure 7). If both values (number of connections and buffer fill level) are above a certain threshold, ECalls reduces the buffer length of the *incomplete connection queue* (SYN-queue in Figure 7), until either the number of connections or the number of accepted requests drop under their respective thresholds. The reason why we use two criteria is to prevent ECalls from decreasing the queue size in case of transient overloads. As an example, if the ACK-queue is above the threshold but the number of open connections is under its threshold, we assume that the server will soon be able to service this burst of requests. On the other hand, if the number of connections is over the threshold, but the buffer fill level is under its threshold, we assume that the small number of pending requests will reduce the server load soon such that it is not necessary to drop requests.

Figures 8(a) and 8(b) show the results of this experiment. Using ECalls, we are able to improve the overload case such that more replies are generated and at the same time the average response times are reduced. As an example, Figure 8(a) shows that thttpd is able to service 80 requests per second at

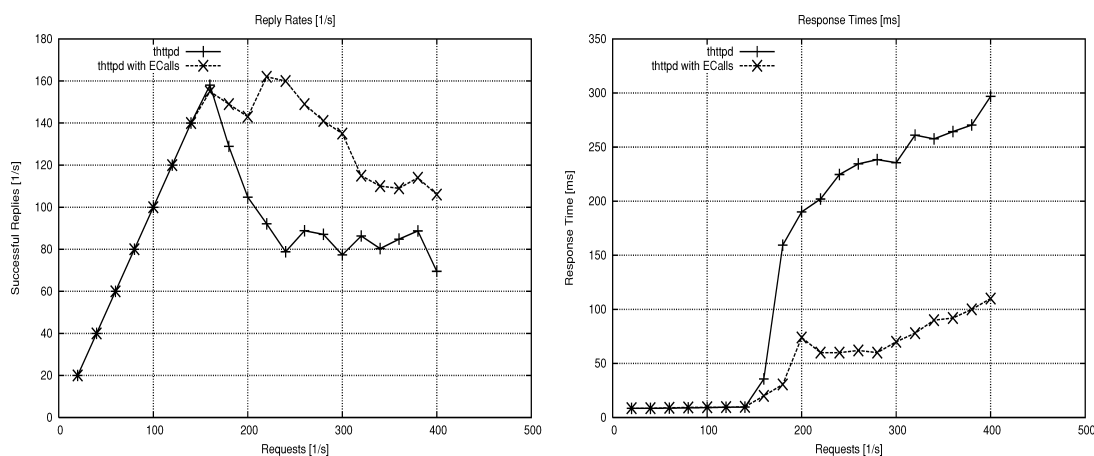


Fig. 8. This graph shows (a) the improved overload behavior of the web server when ECalls modifies the size of the *incomplete connection queue* of the listening socket according to the load and (b) how this adaptive approach also improves the response time of requests.

a request rate of 300 per second, while thttpd using ECalls is able to service 140 requests per second. Figure 8(b) indicates a 250% better response times than the original thttpd Web server. The values for the thresholds have been determined via experimentation, and could also be made adaptive instead of fixed as done in this experiment. Also, more knowledge from both kernel-level and user-level can help to improve the overload case even more (such as the average response time measured in the Web server or the type of a request).

5.4 Experiments with a Distributed Video Player

The following results demonstrate ECalls' ability to couple CPU scheduling with event delivery, improving the user-perceived quality of a video player by increasing its attainable frame rate. Linking event delivery (where here the event is the availability of the next video frame) with CPU scheduling ensures that video players are woken up and scheduled exactly when it is time to display the next image, resulting in an event-aware CPU scheduling approach. Specifically, we modify a distributed MPEG video player such that it uses ECalls to communicate with the I/O module described in Section 5.2. A number of video players (running on a Pentium II with 450 MHz and 512-MB RAM) request video streams from several video servers (running on five Ultra 30 s with 248 MHz and 128-MB RAM each). Each video player writes the *desired frame rate*, a *frame counter*, and the *time stamp* of the last displayed frame into the pinned memory supplied by ECalls. The frame rate can be changed dynamically (e.g., as image resolution or compression changes). The I/O module uses this information to compute the display time of the next frame. Incoming frames are monitored by the I/O module, and ECalls places the notification events into the event queue ordered by the display time of the next frame. The DWCS CPU scheduler uses this information to modify the scheduling priority of the video players. In this experiment, all players have the same attributes of $x/y = 1/5$ and service time $C = 10$ ms, and a priority T corresponding to the desired frame rate, as exemplified by $T = 100$ ms for a frame rate of 10 frames per seconds. The experiment shows that the interaction between an application and a kernel-level service (using ECalls) allows the application to achieve its desired QoS, even when the host is perturbed by several CPU-intensive tasks. Specifically, Figure 9(a) shows that the achieved frame rates drop rapidly when the number of players increases. Using ECalls, we can maintain rates close to the desired frame rates (see Figure 9(b)). This is achieved by delaying event notification for a period of time determined by the

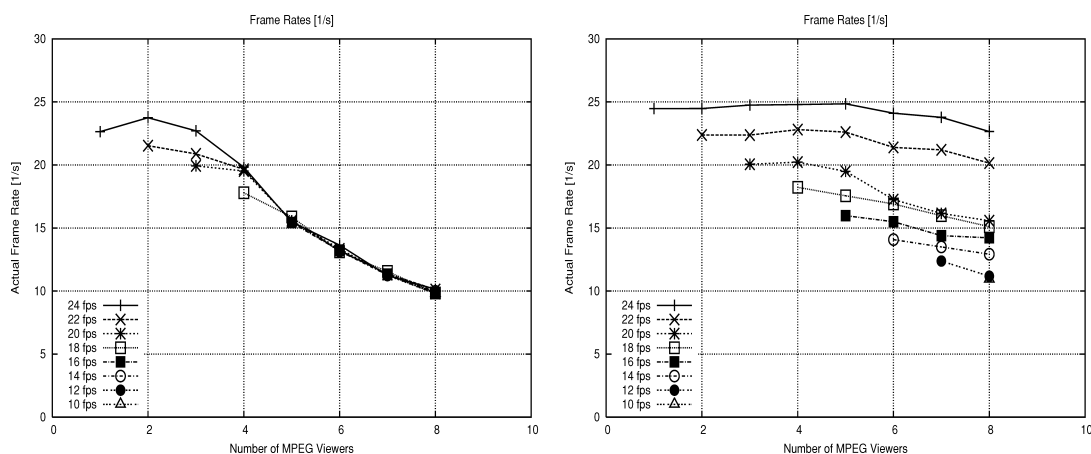


Fig. 9. Achieved frame rates (a) without ECalls and (b) with ECalls.

frame rate, and by influencing scheduling decisions so that the scheduler reorders the run queue to favor applications receiving these events.

6. CONCLUSIONS AND FUTURE WORK

The support of QoS for multimedia applications requires the online quality management from the underlying OS. ECalls rectifies the fact that existing interfaces between applications and system-level services tend to be restrictive and expensive. ECalls supports the efficient communication across protection boundaries in a variety of ways: (1) communication is based on events, where deadlines can be associated with events; (2) besides standard call mechanisms like signals, ECalls uses shared memory between applications and kernel services for low-cost event notification and data sharing; (3) ECalls supports the linking of event dispatching with CPU scheduling, therefore maximizing the responsiveness of applications to critical events, and (4) multiple event-handling approaches can be combined in order to support “filters” or “optimistic” event handlers. An important attribute of ECalls is the ability to dynamically install and use kernel-level event handlers to customize system behavior. Such handlers can be dynamically generated and inserted into a running kernel by an application, both in local and remote systems. Their dynamic generation is based on the run-time conversion of functions written in E-code to machine code within the kernel, resulting in small overheads for event handling.

Experimental evaluations shown in this article address applications like video replay and Web servers. When using ECalls, application behavior and performance are improved substantially. For example, by replacing the *select()* system call in a server with shared memory, ECalls-based implementation of data and control transfer between application and kernel, an increase of 50% is attained for successfully serviced requests in overload situations. The performance improvements achievable with ECalls depend significantly on the cross-domain communication used in existing systems, for example, the *select()* system call approach for Web servers is known to scale poorly with increasing number of requests. Other applications may already use more efficient methods of communication (e.g., zero-copy data movement), that is, the achievable performance improvements depend on the communication mechanisms used, the frequency of communication, and the current system load (e.g., the higher the system load the more significant the effect of event-aware CPU scheduling).

ECalls is implemented as a dynamically loadable kernel module in Linux, but its mechanisms (e.g., use of shared memory, event scheduling) could easily be ported and used in other operating systems. While the event scheduling part of ECalls increases the responsiveness of applications to events on Linux systems using the DWCS CPU scheduler, other CPU schedulers could also be used, although they would require adjustments to the event scheduling rules.

Future work will utilize ECalls as an elementary communication module between protection domains and distributed kernels for “self-managing” computer systems. The goal is to build self-awareness into existing systems, allowing them to monitor, analyze, configure, and steer system-level operations dynamically, thereby maximizing the achievable qualities for applications. A flexible tool for event sharing between domains is essential for satisfying the varying needs of different applications or system-level self-awareness techniques. For example, *self-healing* refers to a system’s capability to react and adjust to failures of system components. Applications have to be informed of failures and a system’s measures to counter these failures in order to effect the self-healing process, examples being to allow applications to “morph” their behaviors to less resource-intensive variants, to off-load tasks to other hosts, or to modify data content dynamically to reflect the change in resource availability. Another potential future direction could address the linkage of ECalls with other, similar research contributions addressing the problems encountered in cross-domain communication. For example, ECalls directly affects the scheduling decisions of the CPU scheduler, but could also profit from an approach such as scheduler activations [Anderson et al. 1991], where upcalls are sent to an application’s thread scheduler.

REFERENCES

- ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. 1991. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*. 95–109.
- BANGA, G., MOGUL, J., AND DRUSCHEL, P. 1999. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference* (Monterey, CA). 253–265.
- BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles* (Cooper Mountain Resort, CO). 267–284.
- BIHARI, T. E. AND SCHWAN, K. 1991. Dynamic adaptation of real-time software. *ACM Trans. Comput. Syst.* 9, 2 (May), 143–174.
- BIRMAN, K. P., VAN RENESSE, R., KAUFMAN, J., AND VOGELS, W. 2003. Navigating in the storm: Using astrolabe for distributed self-configuration, monitoring and adaptation. In *Proceedings of the 5th Annual International Workshop on Active Middleware Services* (Seattle, WA). 4–13.
- CHANDRA, A. AND MOSBERGER, D. 2001. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the USENIX Annual Technical Conference* (Boston, MA). 231–244.
- CLARK, D. 1985. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA). 171–180.
- DRUSCHEL, P. AND PETERSON, L. 1993. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the 14th ACM Symposium of Operating Systems Principles* (Asheville, NC). 189–202.
- EISENHAEUER, G., BUSTAMANTE, F. E., AND SCHWAN, K. 2001. A middleware toolkit for client-initiated service specialization. *ACM SIGOPS 35*, 2 (Apr.), 7–20.
- ENGLER, D. R., KAASHOEK, F. M., AND O’TOOLE, J., JR. 1995. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles* (Cooper Mountain Resort, CO). 251–266.
- GANEV, I., SCHWAN, K., AND EISENHAEUER, G. 2004. Kernel plugins: When a VM is too much. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium* (San Jose, CA). 83–96.
- GOPALAKRISHNAN, R. AND PARULKAR, G. 1998. Efficient user space protocol implementations with QoS guarantees using real-time upcalls. *IEEE/ACM Trans. Netw.* 6, 4, 374–388.
- LEE, C., YOSHIDA, K., MERCER, C., AND RAJKUMAR, R. 1996. Predictable communication protocol processing in real-time mach. In *Proceedings of the Real-time Technology and Applications Symposium* (Boston, MA). 220–229.
- LEMON, J. 2001. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference* (Boston, MA). 141–154.

- MANIMARAN, G., SHASHIDHAR, M., MANIKUTTY, A., AND MURTHY, C. S. R. 1998. Integrated scheduling of tasks and messages in distributed real-time systems. *J. Parall. Distrib. Comput. Pract.* 1, 2 (June), 75–84.
- MOGUL, J. AND RAMAKRISHNAN, K. 1997. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.* 15, 3 (Jan.), 217–252.
- NIEH, J., HANKO, J. G., NORTHCUIT, J. D., AND WALL, G. A. 1993. SVR4 UNIX scheduler unacceptable for multimedia applications. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video* (Lancaster, U.K.). 41–53.
- NIEH, J. AND LAM, M. 2003. A SMART scheduler for multimedia applications. *ACM Trans. Comput. Syst.* 21, 2 (May), 117–163.
- PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999a. Flash: An efficient and portable Web server. In *Proceedings of the USENIX Technical Conference* (Monterey, CA). 199–212.
- PAI, V., DRUSCHEL, P., AND ZWAENEPOEL, W. 1999b. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation* (New Orleans, LA). 37–66.
- PIETZUCH, P. R. AND BACON, J. M. 2002. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops* (Vienna, Austria). 611–618.
- POELLABAUER, C., ABBASI, H., AND SCHWAN, K. 2002. Cooperative run-time management of adaptive applications and distributed resources. In *Proceedings of the 10th ACM Multimedia Conference* (Juan-les-Pins, France). 402–411.
- POLETO, M., ENGLER, D., AND KAASHOEK, M. F. 1996. tcc: A template-based compiler for 'C. In *Proceedings of the First Workshop on Compiler Support for System Software* (Tucson, AZ). 1–7.
- PROVOS, N. AND LEVER, C. 2000. Scalable network I/O in linux. In *FREENIX Track of the Proceedings of the USENIX Annual Technical Conference* (San Diego, CA). 109–120.
- PROVOS, N., LEVER, C., AND TWEEDIE, S. 2000. Analyzing the overload behavior of a simple Web server. In *Proceedings of the 4th Annual Linux Showcase and Conference* (Atlanta, GA). 1–12.
- ROSU, D., SCHWAN, K., YALAMANCHILI, S., AND JHA, R. 1997. On adaptive resource allocation for complex real-time applications. In *Proceedings of the 18th Real-Time Systems Symposium* (San Francisco, CA). 320–329.
- ROSU, M.-C. AND ROSU, D. 2003. Kernel support for faster Web proxies. In *Proceedings of the USENIX Annual Technical Conference* (San Antonio, TX). 225–238.
- STEEER, D., GOEL, A., GRUENBERG, J., MCNAMEE, D., PU, C., AND WALPOLE, J. 1999. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (New Orleans, LA). 145–158.
- WALLACH, D. A., HSIEH, W. C., JOHNSON, K. L., KAASHOEK, F. M., AND WEIHL, W. E. 1995. Optimistic active messages: A mechanism for scheduling communication with computation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Santa Barbara, CA). 217–226.
- WEST, R. AND POELLABAUER, C. 2000. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *Proceedings of the 21st Real-Time Systems Symposium* (Orlando, FL). 239–248.
- YAU, D. AND LAM, S. 1996. An architecture towards efficient OS support for distributed multimedia. In *Proceedings IS&T/SPIE Multimedia Computing and Networking Conference* (San Jose, CA).

Received June 2004; revised November 2004, January 2005; accepted January 2005